Valantis HOT Audit Report

Mar 24, 2024

Table of Contents

Summary

This report has been prepared for Valantis HOT smart contract, to discover issues and vulnerabilities in the source code of their Smart Contract as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

Overview

Project Summary

Project Name	**Valantis HOT**
Codebase	**https://github.com/ValantisLabs/valantis-hot**
Commit	**27d09981a8cf67bdd4032f4018946e21296e2e7e**
Language	**Solidity**

Audit Summary

Delivery Date	**Mar 24, 2024**
Audit Methodology	**Static Analysis, Manual Review**
Total Isssues	**6**

[WP-O1] The signer acts as a single point of failure for the entire system.

Issue Description

There is already a restriction on the deviation of the quote price and the oracle's price, e.g., 1%. However, a malfunctioning/compromised signer can still sign quotes in a certain way and drain the entire pool's funds in a short period of time.

For instance:

Given:

A SOT pool with $10,000 worth of ETH and 10,000 USDC.

The signer can:

1. Sell all the ETH at a 1% discounted price for 9,900 USDC;
2. Sell all the 19,900 USDC at a 1% discounted price for $19,701 worth of ETH;
3. Sell all the ETH at a 1% discounted price for 19,503 USDC;

...

1. Sell all the 11,682 USDC at a 1% discounted price for $11,565 worth of ETH;

Repeat the above steps and trade the entire pool's funds back and forth, each time can leak up to the price deviation.

Using all the 56 quoted swaps in one block (if `maxAllowedQuotes == 56`), the signer can drain up to 43% (0.99**56 == 0.5696) of the pool's funds within one block.

https://github.com/ValantisLabs/valantis-sot/blob/27d09981a8cf67bdd4032f4018946e21296e2e7e/src/SOT.sol#L805-L916

```
805   function _solverSwap(
806       ALMLiquidityQuoteInput memory almLiquidityQuoteInput,
807       bytes memory externalContext,
808       ALMLiquidityQuote memory liquidityQuote
809   ) internal {
```

```
@@ 810,842 @@

843
844     // Pick the discounted or base price, depending on eligibility criteria set above
845     // No need to check one against the other at this stage
846     uint256 solverPriceX192 = isDiscountedSolver ? sot.solverPriceX192Discounted : sot.solverPriceX192Base;
847
848     // Calculate the amountOut according to the quoted price
849     liquidityQuote.amountOut = almLiquidityQuoteInput.isZeroToOne
850         ? Math.mulDiv(almLiquidityQuoteInput.amountInMinusFee, solverPriceX192, SOTConstants.Q192)
851         : Math.mulDiv(almLiquidityQuoteInput.amountInMinusFee, SOTConstants.Q192, solverPriceX192);
852     // Fill tokenIn amount requested, excluding fees
853     liquidityQuote.amountInFilled = almLiquidityQuoteInput.amountInMinusFee;
854
855     // Check validity of new AMM dynamic fee parameters
856     sot.validateFeeParams(minAMMFee, minAMMFeeGrowthInPips, maxAMMFeeGrowthInPips);
857
858     sot.validateBasicParams(
859         liquidityQuote.amountOut,
860         almLiquidityQuoteInput.sender,
861         almLiquidityQuoteInput.recipient,
862         almLiquidityQuoteInput.amountInMinusFee,
863         almLiquidityQuoteInput.isZeroToOne ? maxToken1VolumeToQuote : maxToken0VolumeToQuote,
864         maxDelay,
865         solverWriteSlotCache.alternatingNonceBitmap
866     );
867
868     SOTParams.validatePriceConsistency(
869         _ammState,
870         solverPriceX192.sqrt().toUint160(),
871         sot.sqrtSpotPriceX96New,
872         getSqrtOraclePriceX96(),
873         solverReadSlot.maxOracleDeviationBips,
874         solverMaxDiscountBips
875     );
876
```

```
@@ 877,915 @@
916  }
```

Status

ⓘ Acknowledged

[WP-I2] Using timestamp comparison to check if the last processed quote is in the same block is unreliable on L2s like Arbitrum or L1s with sub-second block times.

Informational

Issue Description

https://github.com/ValantisLabs/valantis-sot/blob/27d09981a8cf67bdd4032f4018946e21296e2e7e/src/SOT.sol#L835-L842

```
835         // Ensure that the number of SOT swaps per block does not exceed its maximum bound
836         uint8 quotesInCurrentBlock = block.timestamp > solverWriteSlotCache.lastProcessedQuoteTimestamp
837             ? 1
838             : solverWriteSlotCache.lastProcessedBlockQuoteCount + 1;
839
840         if (quotesInCurrentBlock > solverReadSlotCache.maxAllowedQuotes) {
841             revert SOT___solverSwap_maxSolverQuotesExceeded();
842         }
```

[WP-I3] `ISovereignPoolSwapCallback.sovereignPoolSwapCallback()` may lack the `sovereignVault` parameter.

Informational

Issue Description

If `ISovereignPoolSwapCallback.sovereignPoolSwapCallback()` includes a `sovereignVault` parameter, it could be more convenient for the caller of `SovereignPool.swap()` to know where to transfer `_tokenIn` when implementing `sovereignPoolSwapCallback(address _tokenIn, uint256 _amountInUsed, bytes calldata _swapCallbackContext)` .

This addition could facilitate a clearer understanding for the implementer about the destination of `_tokenIn` .

https://github.com/ValantisLabs/valantis-core/blob/991850b592b7b5dc444689df0e13f27f3f434ce7/src/pools/SovereignPool.sol#L647-L823

```
@@ 647,661 @@
662     function swap(
663         SovereignPoolSwapParams calldata _swapParams
664     ) external override nonReentrant returns (uint256 amountInUsed, uint256
    amountOut) {
@@ 665,692 @@
693
694         bytes memory verifierData;
695         if (address(_verifierModule) != address(0)) {
696             // Query Verifier Module to authenticate the swap
697             verifierData = _verifyPermission(
698                 msg.sender,
699                 _swapParams.swapContext.verifierContext,
700                 uint8(AccessType.SWAP)
701             );
702         }
703
@@ 704,781 @@
782
783         _handleTokenInTransfersOnSwap(
```

```
784             _swapParams.isZeroToOne,
785             _swapParams.isSwapCallback,
786             swapCache.tokenInPool,
787             amountInUsed,
788             effectiveFee,
789             _swapParams.swapContext.swapCallbackContext
790         );
791
    @@ 792,822 @@
823     }
```

https://github.com/ValantisLabs/valantis-core/blob/991850b592b7b5dc444689df0e13f27f3f434ce7/src/pools/SovereignPool.sol#L1035-L1078

```
1035     function _handleTokenInTransfersOnSwap(
1036         bool isZeroToOne,
1037         bool isSwapCallback,
1038         IERC20 token,
1039         uint256 amountInUsed,
1040         uint256 effectiveFee,
1041         bytes calldata _swapCallbackContext
1042     ) private {
1043         uint256 preBalance = token.balanceOf(sovereignVault);
1044
1045         if (isSwapCallback) {
1046             ISovereignPoolSwapCallback(msg.sender).sovereignPoolSwapCallback(
1047                 address(token),
1048                 amountInUsed,
1049                 _swapCallbackContext
1050             );
1051         } else {
1052             token.safeTransferFrom(msg.sender, sovereignVault, amountInUsed);
1053         }
1054
1055         uint256 amountInReceived = token.balanceOf(sovereignVault) - preBalance;
1056
1057         bool isTokenInRebase = isZeroToOne ? isToken0Rebase : isToken1Rebase;
1058
1059         if (isTokenInRebase) {
1060             uint256 tokenInAbsDiff = amountInUsed > amountInReceived
```

```
1061                 ? amountInUsed - amountInReceived
1062                 : amountInReceived - amountInUsed;
1063
1064             uint256 tokenInAbsErrorTolerance = isZeroToOne ?
     token0AbsErrorTolerance : token1AbsErrorTolerance;
1065             if (tokenInAbsDiff > tokenInAbsErrorTolerance)
1066                 revert
     SovereignPool___handleTokenInOnSwap_excessiveTokenInErrorOnTransfer();
1067         } else {
1068             if (amountInReceived != amountInUsed) revert
     SovereignPool___handleTokenInOnSwap_invalidTokenInAmount();
1069         }
1070
1071         if (isTokenInRebase && sovereignVault == address(this) && poolManager !=
     address(0)) {
1072             // We transfer manager fee to `poolManager`
1073             uint256 poolManagerFee = Math.mulDiv(effectiveFee, poolManagerFeeBips,
     _FACTOR_ONE);
1074             if (poolManagerFee > 0) {
1075                 token.safeTransfer(poolManager, poolManagerFee);
1076             }
1077         }
1078     }
```

https://github.com/ValantisLabs/valantis-core/blob/991850b592b7b5dc444689df0e13f27f3f434ce7/src/pools/interfaces/ISovereignPoolSwapCallback.sol#L4-L10

```
4  interface ISovereignPoolSwapCallback {
5      function sovereignPoolSwapCallback(
6          address _tokenIn,
7          uint256 _amountInUsed,
8          bytes calldata _swapCallbackContext
9      ) external;
10 }
```

[WP-I4] `flashLoan` Recommendation to Adhere to ERC-3156 Standard by Adding Fee Parameter to Callback

Informational

Issue Description

ERC-3156: Flash Loans

> The `flashLoan` function MUST include a fee argument to onFlashLoan with the fee to pay for the loan on top of the principal, ensuring that `fee == flashFee(token, amount)` .

It is suggested to add a fee parameter to the `receiver.onFlashLoan()` callback to maintain consistency with the standard callback function signature.

https://github.com/ValantisLabs/valantis-core/blob/991850b592b7b5dc444689df0e13f27f3f434ce7/src/pools/SovereignPool.sol#L567-L600

```
567       function flashLoan(
568           bool _isTokenZero,
569           IFlashBorrower _receiver,
570           uint256 _amount,
571           bytes calldata _data
572       ) external nonReentrant {
573           // We disable flash-loans,
574           // since reserves are not meant to be stored in the pool
575           if (sovereignVault != address(this)) revert
      ValantisPool__flashLoan_flashLoanDisabled();
576
577           IERC20 flashToken = _isTokenZero ? _token0 : _token1;
578           bool isRebaseFlashToken = _isTokenZero ? isToken0Rebase : isToken1Rebase;
579
580           // Flash-loans for rebase tokens are disabled.
581           // Easy to manipulate token reserves would significantly
582           // increase the attack surface for contracts that rely on this pool
583           if (isRebaseFlashToken) {
584               revert ValantisPool__flashLoan_rebaseTokenNotAllowed();
585           }
586
587           uint256 poolPreBalance = flashToken.balanceOf(address(this));
```

```
588
589         flashToken.safeTransfer(address(_receiver), _amount);
590         if (_receiver.onFlashLoan(msg.sender, address(flashToken), _amount, _data)
    != _CALLBACK_SUCCESS) {
591             revert ValantisPool__flashloan_callbackFailed();
592         }
593         flashToken.safeTransferFrom(address(_receiver), address(this), _amount);
594
595         if (flashToken.balanceOf(address(this)) != poolPreBalance) {
596             revert ValantisPool__flashLoan_flashLoanNotRepaid();
597         }
598
599         emit Flashloan(msg.sender, address(_receiver), _amount,
    address(flashToken));
600     }
```

[WP-L5] AMM Swap won't invoking `_updateAMMLiquidity()` could result in reserves that could have served as liquidity not being utilized as such, potentially leading to less efficient use of funds (compared to Univ2).

Low

Issue Description

When a swap earns fees for LPs and ***additionally*** increases reserves (affecting k), it should invoke `_updateAMMLiquidity()` to continue utilizing the fees for trading.

The way Valantis handles LP fees appears to differ from Uniswap v3:

- Uniswap v3 separates principal and fees, with v3 LP fees not affecting k.
- In Valantis, it seems both principal and fees directly impact the reserve, with the fee portion potentially affecting k.

https://github.com/ValantisLabs/valantis-core/blob/991850b592b7b5dc444689df0e13f27f3f434ce7/src/pools/SovereignPool.sol#L647-L823

```
647     /**
648         @notice Swap against the ALM Position in this pool.
649         @param _swapParams Struct containing all params.
650                * isSwapCallback If this swap should claim funds using a callback.
651                * isZeroToOne Direction of the swap.
652                * amountIn Input amount to swap.
653                * amountOutMin Minimum output token amount required.
654                * deadline Block timestamp after which the swap is no longer valid.
655                * recipient Recipient address for output token.
656                * swapTokenOut Address of output token.
657                  If `sovereignVault != address(this)` it can be other tokens apart
    from token0 or token1.
658                * swapContext Struct containing ALM's external, Verifier's and Swap
    Callback's context data.
659         @return amountInUsed Amount of input token filled by this swap.
660         @return amountOut Amount of output token provided by this swap.
661      */
662     function swap(
```

```
663         SovereignPoolSwapParams calldata _swapParams
664     ) external override nonReentrant returns (uint256 amountInUsed, uint256
    amountOut) {
    @@ 665,721 @@
722
723         // Since we do not yet know how much of `amountIn` will be filled,
724         // this quantity is calculated in such a way that `msg.sender`
725         // will be charged `feeInBips` of whatever the amount of tokenIn filled
726         // ends up being (see docs for more details)
727         swapCache.amountInWithoutFee = Math.mulDiv(
728             _swapParams.amountIn,
729             _MAX_SWAP_FEE_BIPS,
730             _MAX_SWAP_FEE_BIPS + swapFeeModuleData.feeInBips
731         );
732
733         ALMLiquidityQuote memory liquidityQuote =
    ISovereignALM(alm).getLiquidityQuote(
734             ALMLiquidityQuoteInput({
735                 isZeroToOne: _swapParams.isZeroToOne,
736                 amountInMinusFee: swapCache.amountInWithoutFee,
737                 feeInBips: swapFeeModuleData.feeInBips,
738                 sender: msg.sender,
739                 recipient: _swapParams.recipient,
740                 tokenOutSwap: _swapParams.swapTokenOut
741             }),
742             _swapParams.swapContext.externalContext,
743             verifierData
744         );
745
746         amountOut = liquidityQuote.amountOut;
747
748         if (
749             !_checkLiquidityQuote(
750                 _swapParams.isZeroToOne,
751                 swapCache.amountInWithoutFee,
752                 liquidityQuote.amountInFilled,
753                 amountOut,
754                 _swapParams.amountOutMin
755             )
756         ) {
757             revert SovereignPool__swap_invalidLiquidityQuote();
758         }
```

```
759
760         // If amountOut is 0, we do not transfer any input token
761         if (amountOut == 0) {
762             revert SovereignPool__swap_zeroAmountOut();
763         }
764
765         // Calculate the actual swap fee to be charged in input token
    (`effectiveFee`),
766         // now that we know the tokenIn amount filled
767         uint256 effectiveFee;
768         if (liquidityQuote.amountInFilled != swapCache.amountInWithoutFee) {
769             effectiveFee = Math.mulDiv(
770                 liquidityQuote.amountInFilled,
771                 swapFeeModuleData.feeInBips,
772                 _MAX_SWAP_FEE_BIPS,
773                 Math.Rounding.Up
774             );
775
776             amountInUsed = liquidityQuote.amountInFilled + effectiveFee;
777         } else {
778             // Using above formula in case amountInWithoutFee == amountInFilled
    introduces rounding errors
779             effectiveFee = _swapParams.amountIn - swapCache.amountInWithoutFee;
780             amountInUsed = _swapParams.amountIn;
781         }
782
783         _handleTokenInTransfersOnSwap(
784             _swapParams.isZeroToOne,
785             _swapParams.isSwapCallback,
786             swapCache.tokenInPool,
787             amountInUsed,
788             effectiveFee,
789             _swapParams.swapContext.swapCallbackContext
790         );
791
792         // Update internal state and oracle module.
793         // In case of rebase tokens, `amountInUsed` and `amountOut` might not
    match
794         // the exact balance deltas due to rounding errors.
795         _updatePoolStateOnSwap(_swapParams.isZeroToOne, amountInUsed, amountOut,
    effectiveFee);
796
797         if (
```

```
798             address(_sovereignOracleModule) != address(0) &&
799             _swapParams.swapTokenOut == address(swapCache.tokenOutPool) &&
800             amountInUsed > 0
801         ) {
802             _sovereignOracleModule.writeOracleUpdate(_swapParams.isZeroToOne,
    amountInUsed, effectiveFee, amountOut);
803         }
804 
805         // Transfer `amountOut to recipient
806         _handleTokenOutTransferOnSwap(IERC20(_swapParams.swapTokenOut),
    _swapParams.recipient, amountOut);
807 
808         // Update state for Swap fee module,
809         // only performed if internalContext is non-empty
810         if (
811             address(swapCache.swapFeeModule) != address(0) &&
812             keccak256(swapFeeModuleData.internalContext) != keccak256(new
    bytes(0))
813         ) {
814             swapCache.swapFeeModule.callbackOnSwapEnd(effectiveFee, amountInUsed,
    amountOut, swapFeeModuleData);
815         }
816 
817         // Perform post-swap callback to liquidity module if necessary
818         if (liquidityQuote.isCallbackOnSwap) {
819             ISovereignALM(alm).onSwapCallback(_swapParams.isZeroToOne,
    amountInUsed, amountOut);
820         }
821 
822         emit Swap(msg.sender, _swapParams.isZeroToOne, amountInUsed, effectiveFee,
    amountOut);
823     }
```

https://github.com/ValantisLabs/valantis-core/blob/991850b592b7b5dc444689df0e13f27f3f434ce7/src/pools/SovereignPool.sol#L219-L223

```
219     /**
220         @notice Fraction of swap fees that go into `poolManager`, in bips.
221         @dev Remaining fraction goes to LPs.
222      */
223     uint256 public poolManagerFeeBips;
```

https://github.com/ValantisLabs/valantis-core/blob/991850b592b7b5dc444689df0e13f27f3f434ce7/src/pools/SovereignPool.sol#L1091-L1120

```
1091      function _updatePoolStateOnSwap(
1092          bool isZeroToOne,
1093          uint256 amountInUsed,
1094          uint256 amountOut,
1095          uint256 effectiveFee
1096      ) private {
1097          if (isZeroToOne) {
1098              if (!isToken0Rebase) {
1099                  uint256 poolManagerFee = Math.mulDiv(effectiveFee, poolManagerFeeBips, _FACTOR_ONE);
1100
1101                  if (sovereignVault == address(this)) _reserve0 += (amountInUsed - poolManagerFee);
1102                  if (poolManagerFee > 0) feePoolManager0 += poolManagerFee;
1103              }
1104
1105              if (sovereignVault == address(this) && !isToken1Rebase) {
1106                  _reserve1 -= amountOut;
1107              }
1108          } else {
1109              if (sovereignVault == address(this) && !isToken0Rebase) {
1110                  _reserve0 -= amountOut;
1111              }
1112
1113              if (!isToken1Rebase) {
1114                  uint256 poolManagerFee = Math.mulDiv(effectiveFee, poolManagerFeeBips, _FACTOR_ONE);
1115
1116                  if (sovereignVault == address(this)) _reserve1 += (amountInUsed - poolManagerFee);
1117                  if (poolManagerFee > 0) feePoolManager1 += poolManagerFee;
1118              }
1119          }
1120      }
```

Note: `amountInUsed - poolManagerFee` includes the principal amount (maintaining $k = token_0 \cdot token_1$ constant, resulting in $\Delta token_{in}$) and the fee portion.

https://github.com/ValantisLabs/valantis-sot/blob/1fc114901412894b4dd8da7947b6f91acd693858/src/SOT.sol#L531-L557

```
531     // @audit Verify that we don't need a reentrancy guard for
    getLiquidityQuote/deposit/withdraw
532     /**
533         @notice Sovereign ALM function to be called on every swap.
534         @param _almLiquidityQuoteInput Contains fundamental information about the
    swap and `pool`.
535         @param _externalContext Bytes encoded calldata, containing required
    off-chain data.
536         @return liquidityQuote Returns a quote to authorize `pool` to execute the
    swap.
537      */
538     function getLiquidityQuote(
539         ALMLiquidityQuoteInput memory _almLiquidityQuoteInput,
540         bytes calldata _externalContext,
541         bytes calldata /*_verifierData*/
542     ) external override onlyPool onlyUnpaused returns (ALMLiquidityQuote memory
    liquidityQuote) {
543         if (_externalContext.length == 0) {
544             // AMM Swap
545             _ammSwap(_almLiquidityQuoteInput, liquidityQuote);
546         } else {
547             // Solver Swap
548             _solverSwap(_almLiquidityQuoteInput, _externalContext,
    liquidityQuote);
549
550             // Solver swap needs a swap callback, to update AMM liquidity
    correctly
551             liquidityQuote.isCallbackOnSwap = true;
552         }
553
554         if (liquidityQuote.amountOut == 0) {
555             revert SOT__getLiquidityQuote_zeroAmountOut();
556         }
557     }
```

https://github.com/ValantisLabs/valantis-sot/blob/1fc114901412894b4dd8da7947b6f91acd693858/src/SOT.sol#L722-L735

```
722     /**
723         @notice Sovereign Pool callback on `swap`.
724         @dev This is called at the end of each swap, to allow SOT to perform
725              relevant state updates.
726         @dev Only callable by `pool`.
727      */
728     function onSwapCallback(
729         bool /*_isZeroToOne*/,
730         uint256 /*_amountIn*/,
731         uint256 /*_amountOut*/
732     ) external override onlyPool {
733         // Update AMM liquidity at the end of the swap
734         _updateAMMLiquidity();
735     }
```

https://github.com/ValantisLabs/valantis-sot/blob/1fc114901412894b4dd8da7947b6f91acd693858/src/SOT.sol#L923-L955

```
923     /**
924         @notice Helper function to update AMM's effective liquidity.
925      */
926     function _updateAMMLiquidity() private returns (uint128 updatedLiquidity) {
@@ 927,949 @@
950
951         // Update effective AMM liquidity
952         _effectiveAMMLiquidity = updatedLiquidity;
953
954         emit EffectiveAMMLiquidityUpdate(updatedLiquidity);
955     }
```

In the case of AMM Swap, `liquidityQuote.isCallbackOnSwap = true;` is not set.

Only in the case of Solver Swap, `liquidityQuote.isCallbackOnSwap = true;` is set, which then triggers `_updateAMMLiquidity()` during `ISovereignALM(alm).onSwapCallback(_swapParams.isZeroToOne, amountInUsed, amountOut);` .

[WP-M7] The `sqrtPriceX96` in UniswapV3 does not increase linearly, thus it cannot be directly multiplied by a fixed ratio for comparison.

Medium

Issue Description

The traditional method for calculating the difference in linear prices is: `(Price1 - Price0 / Price0) < MaxBPS` .
When translated to the `sqrtPriceX96` representation, we get `Price0 = (sqrtPriceX96_0 / 2 ** 96) ** 2` , `Price1 = (sqrtPriceX96_1 / 2 ** 96) ** 2` .

Therefore, it follows that `sqrtPriceX96_1 ** 2 - sqrtPriceX96_0 ** 2 < MaxBPS * sqrtPriceX96_0 ** 2`

See:
https://blog.uniswap.org/uniswap-v3-math-primer#relationship-between-tick-and-sqrtprice

https://github.com/ValantisLabs/valantis-sot/blob/1fc114901412894b4dd8da7947b6f91acd693858/src/SOT.sol#L806-L917

```
806    function _solverSwap(
807           ALMLiquidityQuoteInput memory almLiquidityQuoteInput,
808           bytes memory externalContext,
809           ALMLiquidityQuote memory liquidityQuote
810       ) internal {
@@ 811,857 @@
858
@@ 859,867 @@
868
869           SOTParams.validatePriceConsistency(
870               _ammState,
871               solverPriceX192.sqrt().toUint160(),
872               sot.sqrtSpotPriceX96New,
873               getSqrtOraclePriceX96(),
874               solverReadSlot.maxOracleDeviationBips,
875               solverMaxDiscountBips
```

```
876         );
877
@@ 878,916 @@
917     }
```

https://github.com/ValantisLabs/valantis-sot/blob/1fc114901412894b4dd8da7947b6f91acd693858/src/libraries/SOTParams.sol#L103-L143

```
103 function validatePriceConsistency(
104     AMMState storage ammState,
105     uint160 sqrtSolverPriceX96,
106     uint160 sqrtSpotPriceNewX96,
107     uint160 sqrtOraclePriceX96,
108     uint256 maxOracleDeviationBips,
109     uint256 solverMaxDiscountBips
110 ) internal view {
111     // Cache sqrt spot price, lower bound, and upper bound
112     (uint160 sqrtSpotPriceX96, uint160 sqrtPriceLowX96, uint160 sqrtPriceHighX96)
    = ammState.getState();
113
114     // sqrt solver and new AMM spot price cannot differ beyond allowed bounds
115     uint256 solverAndSpotPriceNewAbsDiff = sqrtSolverPriceX96 >
    sqrtSpotPriceNewX96
116         ? sqrtSolverPriceX96 - sqrtSpotPriceNewX96
117         : sqrtSpotPriceNewX96 - sqrtSolverPriceX96;
118
119     if (solverAndSpotPriceNewAbsDiff * SOTConstants.BIPS > solverMaxDiscountBips *
    sqrtSpotPriceNewX96) {
120         revert
    SOTParams__validatePriceConsistency_solverAndSpotPriceNewExcessiveDeviation();
121     }
122
123     // Current AMM sqrt spot price and oracle sqrt price cannot differ beyond
    allowed bounds
124     uint256 spotPriceAndOracleAbsDiff = sqrtSpotPriceX96 > sqrtOraclePriceX96
125         ? sqrtSpotPriceX96 - sqrtOraclePriceX96
126         : sqrtOraclePriceX96 - sqrtSpotPriceX96;
127
128     if (spotPriceAndOracleAbsDiff * SOTConstants.BIPS > maxOracleDeviationBips *
    sqrtOraclePriceX96) {
```

```
129         revert SOTParams__validatePriceConsistency_spotAndOraclePricesExcessiveDeviation();
130     }
131
132     // New AMM sqrt spot price (provided by SOT quote) and oracle sqrt price cannot differ
133     // beyond allowed bounds
134     uint256 spotPriceNewAndOracleAbsDiff = sqrtSpotPriceNewX96 > sqrtOraclePriceX96
135         ? sqrtSpotPriceNewX96 - sqrtOraclePriceX96
136         : sqrtOraclePriceX96 - sqrtSpotPriceNewX96;
137
138     if (spotPriceNewAndOracleAbsDiff * SOTConstants.BIPS > maxOracleDeviationBips * sqrtOraclePriceX96) {
139         revert SOTParams__validatePriceConsistency_newSpotAndOraclePricesExcessiveDeviation();
140     }
141
142     validatePriceBounds(sqrtSpotPriceNewX96, sqrtPriceLowX96, sqrtPriceHighX96);
143 }
```

https://github.com/ValantisLabs/valantis-sot/blob/1fc114901412894b4dd8da7947b6f91acd693858/src/SOT.sol#L572-L609

```
572  function depositLiquidity(
573         uint256 _amount0,
574         uint256 _amount1,
575         uint160 _expectedSqrtSpotPriceLowerX96,
576         uint160 _expectedSqrtSpotPriceUpperX96
577     ) external onlyLiquidityProvider onlyUnpaused returns (uint256 amount0Deposited, uint256 amount1Deposited) {
578         // Allow `liquidityProvider` to cross-check sqrt spot price against expected bounds,
579         // to protect against its manipulation
580         uint160 sqrtSpotPriceX96Cache = _checkSpotPriceRange(
581             _expectedSqrtSpotPriceLowerX96,
582             _expectedSqrtSpotPriceUpperX96
583         );
584
585         uint160 sqrtOraclePriceX96 = getSqrtOraclePriceX96();
586
```

```
587         // Current AMM sqrt spot price and oracle sqrt price cannot differ beyond
    allowed bounds
588         uint256 spotPriceAndOracleAbsDiff = sqrtSpotPriceX96Cache >
    sqrtOraclePriceX96
589             ? sqrtSpotPriceX96Cache - sqrtOraclePriceX96
590             : sqrtOraclePriceX96 - sqrtSpotPriceX96Cache;
591
592         if (
593             spotPriceAndOracleAbsDiff * SOTConstants.BIPS >
    solverReadSlot.maxOracleDeviationBips * sqrtOraclePriceX96
594         ) {
595             revert SOT__depositLiquidity_spotPriceAndOracleDeviation();
596         }
597
598         // Deposit amount(s) into pool
599         (amount0Deposited, amount1Deposited) =
    ISovereignPool(pool).depositLiquidity(
600             _amount0,
601             _amount1,
602             liquidityProvider,
603             '',
604             ''
605         );
606
607         // Update AMM liquidity with post-deposit reserves
608         _updateAMMLiquidity();
609     }
```

Recommendation

Change to:

```
uint256 sqrtSpotPrice = sqrtSpotPriceX96Cache * sqrtSpotPriceX96Cache;
uint256 sqrtOraclePrice = sqrtOraclePriceX96 * sqrtOraclePriceX96;
uint256 spotPriceAndOracleAbsDiff = sqrtSpotPrice > sqrtOraclePrice
            ? sqrtSpotPrice - sqrtOraclePrice
            : sqrtOraclePrice - sqrtSpotPrice;

if (
    spotPriceAndOracleAbsDiff * SOTConstants.BIPS >
solverReadSlot.maxOracleDeviationBips * sqrtOraclePrice
```

```
) {
    revert SOT__depositLiquidity_spotPriceAndOracleDeviation();
}
```

Status

✓ Fixed

Appendix

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by WatchPug; however, WatchPug does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication.

Disclaimer

This report is based on the scope of materials and documentation provided for a limited review at the time provided. Results may not be complete nor inclusive of all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Smart Contract technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. A report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.